An Intrusion Detection System for Aglets

Giovanni Vigna, Bryan Cassell, and Dave Fayram

Department of Computer Science
University of California Santa Barbara
vigna,dfayram,bryanc@cs.ucsb.edu

Abstract. Mobile agent systems provide support for the execution of mobile software components, called agents. Agents acting on behalf of different users can move between execution environments hosted by different organizations. The security implications of this model are evident and these security concerns have been addressed by extending the authentication and access control mechanisms originally conceived for distributed operating systems to mobile agent systems. Other well-known security mechanisms have been neglected. In particular, satisfactory auditing mechanisms have seldom been implemented for mobile agent systems. The lack of complete and reliable auditing makes it difficult to analyze the actions of mobile components to look for evidence of malicious behavior. This paper presents an auditing facility for the Aglets mobile agent system and an intrusion detection system that takes advantage of this facility. The paper describes how auditing was introduced into the Aglets system, the steps involved in developing the intrusion detection system, and the empirical evaluation of the approach.
Keywords: *Mobile Agents, Security, Intrusion Detection, Auditing.*

1 Introduction

Mobile agent systems provide a distributed computing infrastructure that supports the execution of mobile components, called mobile agents [9]. In the most general case, mobile agents act on behalf of different users and, in addition, the nodes that compose the infrastructure may be managed by different authorities (e.g., a university or a private company).

The mobile agent paradigm provides a number of advantages with respect to the traditional client-server paradigm. The ability to relocate the components of an application supports service customization, optimized access to distributed resources, and deployment in a mobile networking environment [25]. On the other hand, the ability to move and execute code fragments has serious security implications [8, 5, 15]. In particular, the recent worm attacks [4, 6] showed that malicious mobile software is tolerant to eradication and allows one to perform distributed denial-of-service attacks.

The security issues introduced by mobile agents have been addressed by extending the authentication and access control mechanisms originally conceived for distributed systems to address mobility [10]. The goal of authentication and access control mechanisms is to prevent impersonation and unauthorized access to system resources. These mechanisms provide little support for the detection of attacks that either circumvent protection mechanisms or abuse legitimate access to resources. These attacks can be detected by analyzing the operating system information generated by the actions performed by users and applications. The process of collecting this information is called auditing [2].

Meaningful and complete auditing information is a prerequisite for effective intrusion detection. If the information included in the collected events is not complete, attacks may go undetected. In addition, intrusion detection may not be possible at all because attacks may not have a manifestation that can be identified in the audit trail [32].

Unfortunately, this is the case for mobile agent systems. Most mobile agent systems provide no auditing mechanisms or are able to produce only incomplete information about the activity of mobile agents. In addition, different agents are usually executed within a single user process that acts as the execution environment (e.g., a Java Virtual Machine). Therefore, the resulting audit trail at the operating system level contains the actions performed by the execution environment as a whole and cannot be "sorted" to associate a subset of the audit records with the execution of a single mobile agent. To overcome this problem, it is necessary to perform auditing at the agent system level, where meaningful information about the actions of each agent can be collected.

This paper describes the design and implementation of a facility for the collection of audit trails in the Aglets system [20] and an intrusion detection system that takes advantage of the auditing information. The auditing system provides extensive information about the activity of mobile agents running within an execution environment. This information is leveraged by an intrusion detection system specifically developed for Aglets. The system allows one to detect attacks that bypass the authorization mechanisms, abuse legitimate access, or violate the security policy of an Aglets server.

The paper is structured as follows. Section 2 presents related work on the topic of auditing and intrusion detection. Section 3 outlines our approach to detect intrusions in the Aglets system. Section 4 describes how auditing was introduced into the Aglets system. Section 5 presents an intrusion detection system for Aglets. Section 6 contains a quantitative evaluation of the auditing systems and shows examples of the use of the intrusion detection system. Section 7 draws some conclusions and outlines future work.

2 Related Work

Intrusion detection is performed by analyzing one or more input event streams, looking for the manifestation of an attack. Traditionally, the input stream is either represented by packets transmitted on a network segment or by the audit records produced by the auditing facility of an operating system. Examples of event streams are the audit records generated by the Solaris Basic Security Module (BSM) [27], traffic logs collected using tcpdump [28], and syslog messages.

Historically detection has been achieved following two different approaches: anomaly detection and misuse detection. Anomaly detection relies on models of the "normal" behavior of a computer system. These models may focus on the users, the applications, or the network. Behavior profiles may be built by performing statistical analysis on historical data [12, 17] or by using rule-based approaches to specify behavior patterns [19, 34, 35, 26]. Anomaly detection compares actual usage patterns against the established profiles to identify abnormal patterns of activity. Misuse detection systems take a complementary approach. The detection tools are equipped with a number of attack descriptions. These descriptions (or "signatures") are matched against the stream of audit data looking for evidence that the modeled attack is occurring [14, 21, 24]. Misuse and anomaly detection both have advantages and disadvantages. Misuse detection systems can perform focused analysis of the audit data and usually they produce few, if any, false positives. At the same time, misuse detection systems can detect only those attacks that have been modeled. Anomaly detection systems have the advantage of being able to detect previously unknown attacks. This advantage is paid for with a large number of false positives and the difficulty of training a system for a very dynamic environment.

Mobile agents have sometimes been advocated as a means to perform intrusion detection in distributed systems [13, 16, 3, 29]. In this context, intrusion detection systems are designed as mobile applications that roam the network to detect attacks and track intruders. The approach described in this paper takes a different perspective. The approach focuses on the detection of attacks against mobile agent systems and, in particular, against Aglets. Detection is achieved by instrumenting the Aglets system to produce events about the activity of mobile agents and then using a misuse detection tool to analyze the event stream. Similar approaches to auditing of mobile code have been proposed for the Anchor Toolkit [22] and to support policy verification for mobile programs [11]. Unfortunately, the former does not provide complete information about agents activity and, to the best of our knowledge, has not been implemented. The latter does not take into account the mobile agent paradigm and focuses on policy specification and access control for code on demand.

3 Architecture

The goal of our research is to develop mechanisms, techniques, and tools to support intrusion detection in the context of mobile agent systems. The approach includes the design and implementation of mechanisms for the collection of complete auditing information about mobile agent execution and the development of an intrusion detection system that uses the collected audit trails to detect attacks against both the mobile agents and the execution environments.

This paper presents the application of the approach to a particular mobile agent system, namely Aglets, version 2.0.2. Aglets is a well-known mobile agent system whose sources have been made available through the SourceForge open-source initiative [1]. In the Aglets system, mobile agents are called "aglets" and

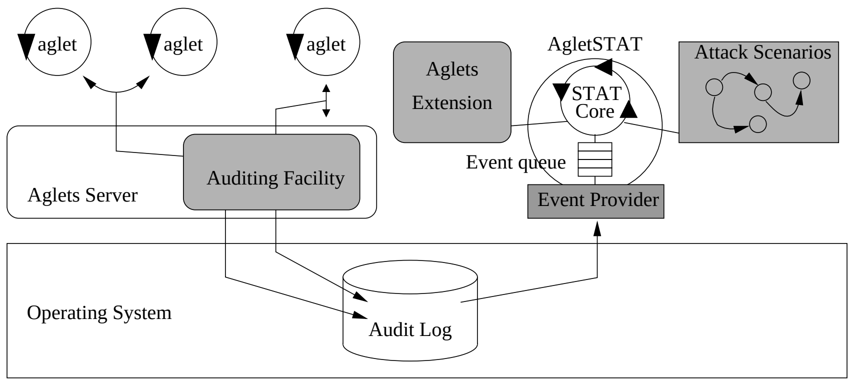


Fig. 1. An intrusion detection system for Aglets.

they are implemented as threads in a Java Virtual Machine, which constitutes the execution environment[1].

The system provides authentication and access control mechanisms [18], but it does not provide an auditing facility that produces information about the activity of mobile agents. To detect attacks performed by aglets, we extended the Aglets system to produce auditing information. In addition, we developed a new intrusion detection system, called AgletSTAT, to analyze the auditing information and identify evidence of malicious behavior. The resulting system allows one to detect attacks against an Aglets server. Figure 1 shows the high-level architecture of the system. The grayed components are the ones that have been developed to implement auditing in Aglets and to build the AgletSTAT intrusion detection system.

The following sections provide details about the process followed to instrument the Aglets system to produce auditing information and describe the AgletSTAT intrusion detection system.

4 Instrumenting the Aglets System

The open-source nature of the Aglets system allowed for the modification of the server's code to intercept the security-relevant actions performed by aglets and log complete information about the identity of an aglet, the requested operation, the parameters of the request, the target of the operation, and the outcome of the request.

The interception of security-relevant operations was achieved by instrumenting the primitives provided by the Aglets system to support agent interaction

[1] In the paper we use the term "Aglets" to denote the Aglets system, the term "Aglets server" to denote the execution environment, and the term "aglet" to denote a mobile agent.

with code that logs the information about the requested operation and the identity of the involved parties. In addition, the Java Security Manager implementation provided with the system was extended to log the parameters of both successful and failed operations.

The aglets interaction procedures that were instrumented are `create`, `clone`, `dispose`, `dispatch`, and `retract`. In addition, the Java Security Manager was modified to log detailed information about system-level operations. For example, logging was implemented for file operations (e.g., `open`, `close`, `read`, and `write`), socket operations (e.g., `accept` and `listen`), and system property operations (for example, the request to modify the `java.home` property).

To log these events, it was necessary to access the information about an executing aglet from its corresponding Java thread. A method was added to the `AgletThread` class. The method returns the aglet's `AgletInfo` structure, which contains all the information necessary to identify the aglet.

The logs produced by the auditing system are in XML format. An audit trail associated with the execution of an Aglets server is an XML document containing a series of events. The structure of an event is designed to accommodate a variety of actions that are relevant in the Aglets system. Each event contains a source, an action, and a result. An example of an `event` element is shown in Figure 2.

```
<event timestamp="Sun Jun 09 15:18:55 PDT 2002"
       server="aglets.mobilecode.net">
   <source>
      <server name="aglets.mobilecode.net" />
   </source>
   <action type="AgletAction create" >
      <target>
         <server name="aglets.mobilecode.net" />
      </target>
   </action>
   <result status="success" type="aglet">
      <aglet id="971045aa51cefd03"
             creationtime="1023661135255"
             class="examples.hello.HelloAglet"
             origin="atp://aglets.mobilecode.net:4434/"
             codebase="atp://home.greetings.com:4434/" />
   </result>
</event>
```

Fig. 2. An example of an event entry in the Aglets audit trail.

The `source` element is used to specify the originator of an action. The originator is either a server (which is usually the local server) or an aglet. The identity of the source is specified in the `name` property.

The action element is used to specify the operation performed by the source. This element contains a type property that defines the type of the action. In the case of actions that are specific to the Aglets system, the type property contains the keyword AgletAction, followed by the name of the operation invoked (e.g., "clone"). In the case of actions involving Java security permissions, the type property contains the name of the permission's class (for example, java.io.FilePermission) followed by the actions associated with the permission (for example, "execute"). A target sub-element is included within the action element if the action requires additional information to specify the object of an operation.

The result element is used to specify the outcome of the operation. This element contains the property status that indicates the success or failure of the action. The property type specifies the type of data enclosed within the result tag. If there is no result data, the result element is empty and the type property is set to "none".

5 An Intrusion Detection System for Aglets

The information generated by the auditing facility of Aglets is used as input to an intrusion detection system, called AgletSTAT. AgletSTAT has been developed by leveraging the STAT framework [33]. The STAT framework provides a platform for the development of intrusion detection sensors by extending a generic runtime with domain-specific components.

The STAT framework is centered around three concepts: the STAT technique, the STATL language, and the STAT Core [31]. The STAT technique is used to represent high-level descriptions of computer attacks. Attack scenarios are abstracted into states, which describe the security status of a system, and transitions, which model the evolution between states.

STATL is an extensible language [7] that is used to represent STAT attack scenarios. The language defines the domain-independent features of the STAT technique. The STATL language can be extended to express the characteristics that are specific to a particular domain. The extension process includes the definition of a set of C++ classes that represent the events in the event stream to be analyzed. In addition, the language is extended with types and predicates that support the definition of events and the testing of domain-specific properties.

Event and predicate definitions are grouped in a language extension module. The module is compiled into a dynamically linked library (i.e., a ".so" file in a UNIX system or a DLL file in a Windows system). Once the event set and associated predicates for a language extension are available, it is possible to use them in a STATL scenario description by including them with the STATL use keyword.

STATL scenarios are matched against a stream of events by the STAT Core. The STAT Core represents the runtime of the STATL language. The STAT Core implements the domain-independent characteristics of STATL, such as the concepts of state, transition, timer, matching of events, etc. At run-time the

STAT Core performs the actual intrusion detection analysis process by matching an incoming stream of events against a number of attack scenarios.

The input event stream is provided by one or more event providers. An event provider collects events from the external environment (e.g., by obtaining packets from the network driver), creates events as defined in one or more STAT language extensions, encapsulates these events into generic STAT events, and inserts these events into the input queue of the STAT core.

In summary, a STAT-based sensor is created by developing a language extension that describes the particular domain of the application, an event provider that retrieves information from the environment and produces STAT events, and attack scenarios that describe attacks in terms of patterns of STAT events.

The AgletSTAT intrusion detection system was developed following the process outlined above. AgletSTAT was built by developing a language extension module that defines Aglets-specific events, an event provider that parses Aglets audit trails and generates Aglets events, and a number of scenarios that detect attacks by analyzing the Aglets event stream.

The AgletSTAT language extension contains the definition of the Aglets event, auxiliary types, and Aglets-specific predicates. Figure 3 shows a simplified version of the class for the Aglets event, which is an abstraction of an entry in the audit log generated by the Aglets auditing facility.

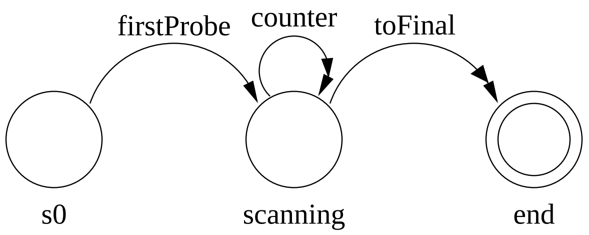
```
  class AgletsEvent : public STATExtEvent {
  public:
    Time timestamp;            // Timestamp
    AgletsComponent source;    // Originator of this event
    AgletsAction action;       // Action performed
    AgletsComponent target;    // Target of this action
    bool outcome;              // Outcome of the operation
    AgeltsActionResult result; // Result of the operation

    [...]
}
```

Fig. 3. The `AgletsEvent` defined in the AgletSTAT language extension.

The AgletSTAT event provider reads the events stored in the audit log file as they are generated. The event provider parses the XML representation of the events and creates the corresponding `AgletsEvent` objects. These objects are encapsulated into STAT events and inserted in the STAT Core event queue. The STAT Core extracts the events from the event queue and passes them to the active attack scenarios for analysis (see Figure 1).

Attacks are represented by specifying state-transition patterns over the stream of `AgletsEvent` events. An example attack scenario is shown in Figure 4. The scenario detects an aglet attempting to perform a portscan against the local



```
use agletstat;

scenario portscan(int threshold)
{
  global HashTable cloneWatch;
  HashTable portWatch;
  string agletid;
  int count = 0;

  initial state s0 {}
  state scanning {}
  state end {
    {stat_log("Aglet %s is performing a portscan", agletid);}
  }
  transition firstProbe (s0->scanning) nonconsuming {
    [AgletsEvent a]:
      (a.action.matches("java.net.SocketPermission connect,resolve") &&
       !(cloneWatch.contains(a.source.id)) &&
       (a.target.name.matches("127.0.0.1")))
    { count++;
      agletid = a.source.id;
      cloneWatch.put(agletid);
      portWatch.put(a.target.port); }
  }
  transition counter (scanning->scanning) consuming {
    [AgletsEvent a]:
      (a.action.matches("java.net.SocketPermission connect,resolve") &&
      (a.source.id == agletid) &&
      (a.target.name.matches("127.0.0.1"))
      (!(portWatch.contains(a.target.port)))
    { count++; }
  }
  transition toFinal (scanning->end) consuming {
    [AgletsEvent a]:
      (count > threshold)
    { cloneWatch.delete(agletid); }
  }
}
```

Fig. 4. An AgletSTAT attack scenario that detects a portscan. The scenario has been simplified with respect to the original, for the sake of exposition. For a detailed description of STATL syntax and semantics see [7]

Aglets server. The attack is detected by counting the number of attempted connections to different ports on the local host and alerting when a predetermined threshold is reached.

A number of STATL scenarios have been developed to detect disclosure of sensitive information (such as accessing the password file on a UNIX system), denial-of-service attacks, and the scanning of remote systems. Note that STATL scenarios can also be used to specify intended aglet behavior. In this case, a scenario contains a state-transition description of the correct sequence of operation to be performed by an aglet. Deviations from the specification would then be detected by the system.

6 Evaluation

The evaluation of the prototype implementation addresses both the auditing system and the intrusion detection tool. The auditing system has been evaluated quantitatively by determining the overhead introduced by the logging procedures. The intrusion detection tool has been evaluated from the functionality point of view by running malicious aglets and analyzing the effectiveness of the detection process.

6.1 The Auditing System

The evaluation of the auditing system was carried out on a system with dual Celeron CPUs clocked at 533 MHz, 384 MB of RAM, and an IBM 7200 rpm hard disk with a DMA-33 interface. The operating system was Linux, kernel version 2.4.16, with Java SDK 1.3.1.

To test the overhead introduced by the auditing system three aglets were developed, namely *CpuBound*, *IOBound*, and *Mixed*. The *CpuBound* aglet performs purely computational operations with doubles, integers, and longs. The *IOBound* aglet is designed to stress-test the auditing system. The aglet simply opens and closes files in a loop, flooding the auditing system with logging requests. The *Mixed* aglet was designed to be a compromise between *CpuBound* and *IOBound*. This aglet opens a file and writes some data to it while doing some computations. Then, the aglet closes the file and opens it again to read the data, while doing more computations. All the aglets have the ability to spawn a specified number of clones of themselves.

Each aglet was run on both the original Aglets system and the instrumented version with 0, 9, and 24 clones, resulting in a total of 1, 10, and 25 aglets running simultaneously. Each test was executed 10 times. The *CpuBound* aglet was run for 50 million iterations with 1 aglet, 10 million iterations per aglet with 10 aglets, and 4 million iterations with 25 aglets. The *IOBound* aglet was run for 20 thousand iterations with 1 aglet, 2 thousand iterations with 10 aglets, and one thousand iterations with 25 aglets. The *Mixed* aglet was run for 3 thousand iterations with 1 aglet, 500 iterations with 10 aglets, and 200 iterations with 25 aglets. The number of iterations for each test was chosen so that all the tests would take are roughly the same time. The setup was identical for each test.

The results of the tests are provided in Table 1. The table contains the execution time required by each test, in milliseconds. An average value is provided for each set of tests. The performance overhead is computed comparing the results for the original system with the results for the instrumented version.

	CpuBound			*IOBound*			*Mixed*		
aglets	1	10	25	1	10	25	1	10	25
Original Aglets									
	63,834	63,120	63,987	24,455	26,265	37,833	51,475	49,267	50,655
	62,046	62,663	63,852	24,793	25,913	36,384	51,834	48,365	50,137
	62,050	62,871	63,862	24,145	25,681	36,936	52,075	48,679	50,453
	62,064	62,859	63,869	24,299	25,938	36,657	51,921	48,600	50,371
	62,104	63,179	64,102	24,453	26,285	36,775	52,458	48,347	50,383
	62,066	62,810	63,916	24,559	26,182	36,160	52,008	48,584	50,830
	62,067	62,820	64,004	24,186	25,898	36,515	51,771	48,749	50,316
	62,047	62,716	64,173	24,536	26,203	36,438	51,881	48,646	50,788
	62,046	63,080	64,298	24,298	25,764	36,375	52,770	49,244	50,743
	62,068	62,789	64,290	24,601	25,958	36,697	51,521	48,531	50,565
Mixed	62,239	62,891	64,035	24,433	26,009	36,677	51,971	48,701	50,524
Modified Aglets									
	63,842	63,491	64,212	72,978	60,845	78,860	61,519	59,221	60,354
	63,854	62,757	63,962	72,625	59,569	78,013	62,771	59,032	59,484
	62,045	62,957	63,844	72,008	59,518	78,558	59,660	58,595	60,013
	62,068	63,012	63,933	71,849	59,603	77,882	59,655	58,717	59,541
	62,086	62,963	64,833	71,540	59,799	78,484	59,228	58,215	59,586
	62,047	62,850	63,983	71,883	59,554	78,254	59,920	58,830	59,334
	62,127	63,015	63,974	71,713	59,675	78,393	60,067	58,305	59,746
	62,098	62,993	64,951	71,892	60,202	78,684	59,705	58,853	60,109
	62,044	63,021	64,525	71,816	59,891	78,992	59,513	58,287	59,885
	62,046	62,996	64,064	71,824	59,836	78,711	59,589	58,521	59,684
Mixed	62,426	63,006	64,228	72,013	59,849	78,483	60,163	58,658	59,774
Overhead									
	0.30%	0.18%	0.30%	194.74%	130.11%	113.98%	15.76%	20.44%	18.31%

Table 1. Performance evaluation of the modified Aglets system.

The impact of auditing is clear in the case of the *IOBound* aglet. This aglet represents the absolute worst possible case for the performance of the auditing system. Every time the *IOBound* aglet accesses a file, the logging system has to log the request, which creates considerable overhead. The worst case shows a three-fold increase in time with respect to the original Aglets system. The *Mixed*[2]

[2] Note that although this aglet is called "*Mixed*", the aglet is still very file-system intensive and generates a large number logging requests.

aglet produced an overhead between 15% and 20%. This overhead is comparable to the overhead introduced by operating system-level auditing facilities [23]. Note that the auditing mechanism introduced into Aglets can be selectively disabled. The tests described here were performed with full logging.

6.2 Evaluating the Intrusion Detection System

The evaluation of the intrusion detection functionality of AgletSTAT was performed by developing a number of malicious aglets and STATL scenarios to detect the attacks.

A first class of malicious aglets perform denial-of-service attacks that attempt to monopolize and/or overload system resources in various ways. The first example is the *CloneBomb* aglet. This aglet generates hundreds of clones of itself. The cloned aglets also generate clones of themselves in a recursive fashion. While inelegant, this attack is extremely effective at monopolizing system resources and proved to be difficult to halt because explicitly disposing of one aglet has no effect. Fortunately, the auditing facility added to the Aglets system logs all aglet-level actions. Therefore, it was possible to develop a scenario to detect excessive cloning. A simple scenario could shut down the server in the case of a clone attack, while a more sophisticated scenario could attempt to explicitly kill all cloning aglets.

Another type of denial-of-service attack proved to be undetectable. In this attack, an aglet, called *WindowBomb*, simply spams the screen with a large number of windows. The aglet uses a collection object to prevent garbage collection of the window objects. Window creation operations cannot be logged properly because the Java security model does not allow for the interception of these events. The *WindowBomb* aglet would be difficult to stop if combined with the *CloneBomb* aglet, although in this case the attack would generate copious audit information and would be easily detected.

A second class of attacks exploits the aglet interaction procedures provided by the Aglets system. An example of this attack is an aglet called *KillBomb* that sends a "dispose" message to every other aglet in the system. The actions necessary to perform this attack are clearly visible in the logs, and a STAT scenario has been developed to detect the attack.

The test of the intrusion detection system also included the execution of attacks that attempt to access security-relevant system properties (e.g., `java.home` or `user.name`) and sensitive files in the operating system (e.g., `/etc/passwd`). These attacks were blocked by the Java Security Manager and logged by the auditing system. STAT scenarios to detect these attempts were developed.

7 Conclusions and Future Work

The goal of our research work is to develop mechanisms and tools to perform intrusion detection in the context of mobile agent systems. Detection of malicious activity performed by the agents is achieved by instrumenting a mobile agent system to generate complete and meaningful auditing information and then analyzing the audit trail using an intrusion detection system.

This paper introduces the general approach and describes its application to the Aglets mobile agent system. The Aglets system was extended to generate auditing information about the actions of mobile agents. In addition, a STAT-based sensor was developed to analyze the audit trail and a number of attack scenarios were developed. The overhead of the auditing system and the effectiveness of detection were evaluated.

Future work will focus on extending this approach in a number of ways. First of all, we will take advantage of the lessons learned in developing our prototype to design a system-independent auditing facility for mobile agent systems. By doing this, a number of mobile agent systems can benefit from the services provided by the auditing mechanism. In order for the auditing facility to be easily included into different mobile agent systems, it is necessary to define both a standardized service interface and a common audit record format. A well-defined common audit trail format would be beneficial to both the intrusion detection and the mobile agent communities. It would support component and preprocessor reuse, simplify data sharing among different systems, and allow for the merging of different event streams.

As a second step, we plan to investigate how intrusion detection can be performed on audit trails collected on different execution environments. Mobile agents can roam from server to server and generate an audit trail at each of the visited hosts. As a consequence, the manifestation of an intrusion may span different hosts or even different mobile agent systems. Therefore, it is necessary to devise a mechanism to reliably collect the audit streams associated with the entire lifetime of an agent. Unfortunately, mobile agent systems may be administered by different authorities with different levels of security and conflicting goals. The collection mechanism should ensure that the audit records associated with the actions of an agent within a server cannot be modified illegally by either the server or the roaming agent.

We plan to build on the cryptographic tracing mechanism introduced in [30]. The mechanism generates a chain of signed checksums of the agent's state and audit trail. The signed checksums are computed at each of the servers visited by an agent. This mechanism makes it impossible for either a server or an agent to tamper with the agent audit trail without being detected. The collected traces are the basis for detecting both attacks against mobile agent systems and attacks against mobile agents.

Acknowledgments

We would like to thank prof. Richard Kemmerer for his comments on early drafts of this work. In addition, we want to thank the anonymous reviewers for their constructive comments that improved the quality of the paper. In particular, we would like to thank one of the reviewers (and give him/her credit) for pointing

out a problem in the tracking of the identities of aglets and for providing a viable solution to the problem.

References

1. The Aglets Software Development Kit (ASDK). http://sourceforge.net/projects/-aglets/, June 2002.
2. J.P. Anderson. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co., Fort Washington, April 1980.
3. J.S. Balasubramaniyan, J.O. Garcia-Fernandez, D. Isacoff, E.H. Spafford, and D. Zamboni. An Architecture for Intrusion Detection Using Autonomous Agents. In *Proceedings of ACSAC '98*, pages 13–24, 1998.
4. CERT/CC. "Code Red Worm" Exploiting Buffer Overflow In IIS Indexing Service DLL. Advisory CA-2001-19, July 2001.
5. D.M. Chess. Security Issues in Mobile Code Systems. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 1–14. Springer-Verlag, June 1998.
6. CIAC. The Ramen Worm. Information Bulletin L-040, February 2001.
7. S.T. Eckmann. *The STATL Attack Detection Language*. PhD thesis, Department of Computer Science, UCSB, Santa Barbara, CA, June 2002.
8. W.M. Farmer, J.D. Guttman, and V. Swarup. Security for Mobile Agents: Issues and Requirements. In *Proc. of the 19th National Information Systems Security Conf.*, pages 591–597, Baltimore, MD, USA, October 1996.
9. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
10. R.S. Gray, D. Kotz, G. Cybenko, and D. Rus. D'Agents: Security in a multiple-language, mobile-agent system. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. Springer-Verlag, 1998.
11. B. Hashii, S. Malabarba, R. Pandey, and M. Bishop. Supporting reconfigurable security policies for mobile programs. *Computer Networks*, 33(1-6):77–93, June 2000.
12. Paul Helman and Gunar Liepins. Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse. In *IEEE Transactions on Software Engineering*, volume Vol 19, No. 9, pages 886–901, 1993.
13. G. Helmer, J.S. K. Wong, V. Honavar, and L. Miller. Intelligent Agents for Intrusion Detection. In *Proceedings of the IEEE Information Technology Conference*, pages 121–124, Syracuse, NY, September 1998.
14. K. Ilgun, R.A. Kemmerer, and P.A. Porras. State Transition Analysis: A Rule-Based Intrusion Detection System. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.
15. W. Jansen and T. Karygiannis. Mobile Agent Security. NIST Special Publication 800-19, August 1999.
16. W. Jansen, P. Mell, T. Karygiannis, and D. Marks. Applying mobile agents to intrusion detection and response. Technical Report 6416, NIST, October 1999.
17. H. S. Javitz and A. Valdes. The NIDES Statistical Component Description and Justification. Technical report, SRI International, Menlo Park, CA, March 1994.
18. G. Karjoth, D. Lange, and M. Oshima. A Security Model for Aglets. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer, 1998.

19. C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 175–187, May 1997.
20. Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley Longman, 1998.
21. U. Lindqvist and P.A. Porras. Detecting Computer and Network Misuse with the Production-Based Expert System Toolset (P-BEST). In *IEEE Symposium on Security and Privacy*, pages 146–161, Oakland, California, May 1999.
22. S. Mudumbai, A. Essiari, and W. Johnston. Anchor Toolkit, 1999.
23. S. Nitzberg. Performance benchmarking of unix system auditing. Master's thesis, Monmouth College, August 1994.
24. M. Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX LISA '99 Conference*, November 1999.
25. Gruia-Catalin Roman, Gian Pietro Picco, and Amy L. Murphy. Software Engineering for Mobility: A Roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*, pages 241–258. ACM Press, 2000.
26. F. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.
27. Sun Microsystems, Inc. *Installing, Administering, and Using the Basic Security Module.* 2550 Garcia Ave., Mountain View, CA 94043, December 1991.
28. Tcpdump and Libpcap Documentation. http://www.tcpdump.org/, June 2002.
29. A. Tripathi, T. Ahmed, S. Pathak, A. Pathak, M. Carney, M. Koka, and P. Dokas. Active Monitoring of Network Systems using Mobile Agents. Technical report, Department of Computer Science, University of Minnesota, May 2002.
30. G. Vigna. Cryptographic Traces for Mobile Agents. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, June 1998.
31. G. Vigna, S. Eckmann, and R. Kemmerer. The STAT Tool Suite. In *Proceedings of DISCEX 2000*, Hilton Head, South Carolina, January 2000. IEEE Computer Society Press.
32. G. Vigna, S.T. Eckmann, and R.A. Kemmerer. Attack Languages. In *Proceedings of the IEEE Information Survivability Workshop*, Boston, MA, October 2000.
33. G. Vigna, R.A. Kemmerer, and P. Blix. Designing a Web of Highly-Configurable Intrusion Detection Sensors. In W. Lee, L. Mè, and A. Wespi, editors, *Proceedings of the 4th International Symposiun on Recent Advances in Intrusion Detection (RAID 2001)*, volume 2212 of *LNCS*, pages 69–84, Davis, CA, October 2001. Springer-Verlag.
34. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001. IEEE Press.
35. C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.